DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Peer-to-Peer-Systems and Security

Gossip: Midterm Report

Gossip-4: Wlad Meixner, Robert Schambach

Contents

1 Introduction

This midterm report documents the following aspects of the Gossip implementation:

- the changes to the Gossip-4 initial report [1].
- the Gossip module's design.
- the module's employed *Peer-to-Peer* (P2P) protocol.

Further, this report illustrates what work the Gossip-4 group must perform to complete the module, as well as how much effort the members required to implement their specific workload. Unless specified otherwise, [2] defines the `MESSAGE TYPES` written in this document.

2 Report Body

2.1 Changes to Initial Report Assumptions

The changes to the assumptions made in the initial report [1] are minimal. These executed changes specifically target the module's design, shown in Section 2.2.1. This section highlights these design changes, as well as noting minor adjustments with regard to other sections of the initial report.

In contrast to the initial report, the logical module structure no longer contains a Validator submodule. As only the Publisher uses this submodule and as this submodule contains little functionality other than checking the well-formed validation bit of `GOSSIP VALIDATION` messages [2], we merged the Validator into the *Application Programming Interface* (API) Communication and Publisher submodule. The API Communication thereby parses the well-formed bit of the `GOSSIP VALIDATION` into a boolean upon deserializing the message itself. Further, the Publisher asserts whether this boolean is set to true. This assertion defines whether the message is well-formed. If this boolean is false, the Publisher returns a Rust error to the caller.

Besides returning an error when a message is not well-formed, the Publisher also errs when sending a message fails. For instance, no subscribers may exist for a given topic. The calling module may then handle this error accordingly.

Due to ease-of-implementation and functional coherence, we moved a partition of the Publisher's logic into the API Communication. This partition pertains to the management of subscriptions. As the active subscriptions correspond to the active API connections, the API Communication submodule implicitly manages the subscriptions in any implementation. Hence, to decrease the code complexity, this submodule explicitly manages the subscriptions. Having the Publisher manage these subscriptions would require additional communication between these submodules, thus increasing complexity. Nevertheless, the Publisher remains as an interface for publishing data to the node's modules, thereby using the API Communication.

The P2P Module has seen the most change compared to the initial report. In this module, we now use TCP instead of UDP sockets to provide a more reliable connection. Having this reliable connection allows us to more easily track the connection status between two peers. This additional state is required for future additions like encryption. TCP guarantees a reliable connection, further simplifying the initial connection process.

Additionally, the P2P serialization library was changed to "PROST!". We tried both options ("rust-protobuf" and "PROST!") and decided that "PROST!" is easy to use and has good type integration with our development tools of choice.
The per-message Proof-of-Work concept was abandoned in favor of a per-connection Proof-Of-Work model. As in a Voice Over IP (VOIP) system the package delays are critical, we found that introducing further PoW caused processing delays on data packets would lead to an unpleasant end-user experience. The connecting party sends out a connection request packet which includes the challenge. Afterwards, the receiving party transmits its own challenge request. Both peers await a valid well-formed connection response, only after which the other peer is considered valid. This process is discussed in further detail in Section 2.3.

2.2 Architecture

This section describes the architecture of the Gossip module with respect to the following aspects:

- the logical structure: what are the Gossip components and how do they interact, in Section 2.2.1.
- the process architecture: how is the execution of the Gossip module structured, in Section 2.2.2.
- the networking model: how does the Gossip module organize its networking capabilities, in Section 2.2.3.

2.2.1 Logical Structure

The Gossip module's overall structure as well as how it interacts with external entities is shown in Figure 2.1. We divide Gossip's internal structure into the following five key submodules: P2P-Communication, API-Communication, Proof-of-Work, Validator, Publisher, and Broadcaster. We refer to [1] for the individual roles of these submodules, as well as to 2.1 for updates regarding these roles. In this section, we focus on the internal and external interactions of these submodules. We thereby separate these interactions into internal and external categories. Internal interactions thereby take place inside an individual node, and nodes perform external interactions with other nodes connected to the P2P network.

Internal Interactions

The API-Communication handles all node internal interaction. The Gossip's Broadcaster and Publisher submodule thereby actively communicate with other submodules.

Regarding outgoing communication, the Broadcaster performs two types of communication: peer-selection requests to the *Random Peer Sampling* (RPS) module, and publish requests to subscribed modules via the Publisher. For peer-selection, the Broadcaster sends the RPS an `RPS QUERY` via the API-Communication submodule. Upon processing the query, the RPS module returns an `RPS PEER` message, which the API-Communication submodule returns to the Broadcaster for further use.

The Broadcaster must also propagate information within the node itself, i.e., to other modules, shown in detail in Figure 2.2 as Publish. To this end, this submodule uses the Publisher to spread data to subscribed modules. The Broadcaster thereby passes the to-be-spread data along with the corresponding data topic to the Publisher. This component parses the received data into a `GOSSIP NOTIFICATION` message, which the Publisher passes further to the API-Communication submodule. This submodule then retrieves and writes the message to the corresponding connections. Upon receiving and processing the `GOSSIP NOTIFICATION`, the modules return a `GOSSIP VALIDATION` message to the Publisher via the API-Communication. Using the validation message, the Publisher returns the result of this transaction to the Broadcaster. Such a result is an error if for instance the validation message signals ill-formed published data or no modules subscribed to the data topic.

Incoming internal communication consists of modules subscribing to and spreading information via the Gossip module. Figure 2.2 again shows these processes as Subscribe and Broadcast, respectively. Modules subscribe to Gossip by sending a `GOSSIP NOTIFY` message to Gossip's config-defined interface. The API-Communication then appends a reference to the connection to the assigned reference list of the contained data topic inside a key-value store. This form of saving the reference allows the API-Communication to retrieve all connection references to a corresponding data topic.

Modules may propagate information to other peers via Gossip. To this end, modules must send a `GOSSIP ANNOUNCE` message to the Gossip module via the API-Communication. Upon parsing the received broadcast message, the API-Communication forwards the parsed message to the Broadcaster submodule for further propagation.

External Interactions

The P2P-Communication submodule handles all external communications between nodes on the P2P network. This submodule exposes an API for connection, direct message transmission and broadcasting. We will use this module in combination with

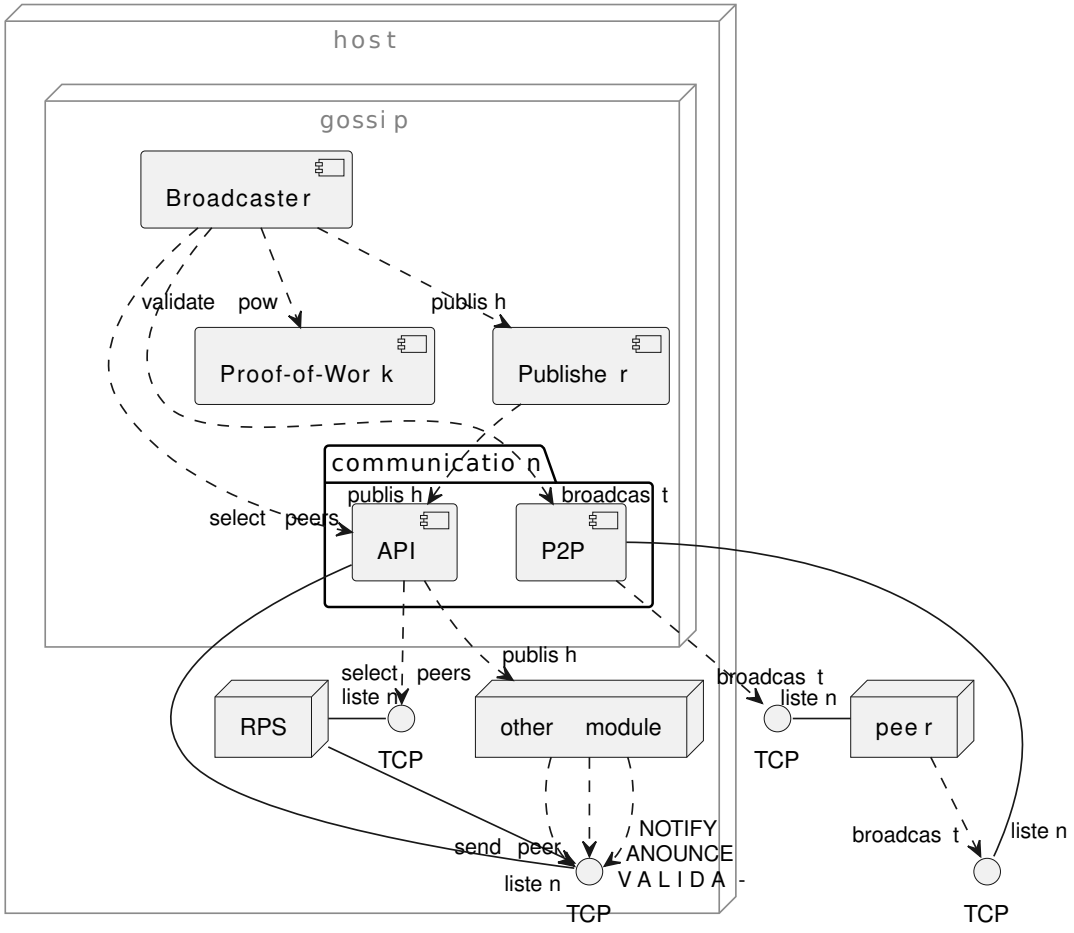

Figure 2.1: Architecture Overview

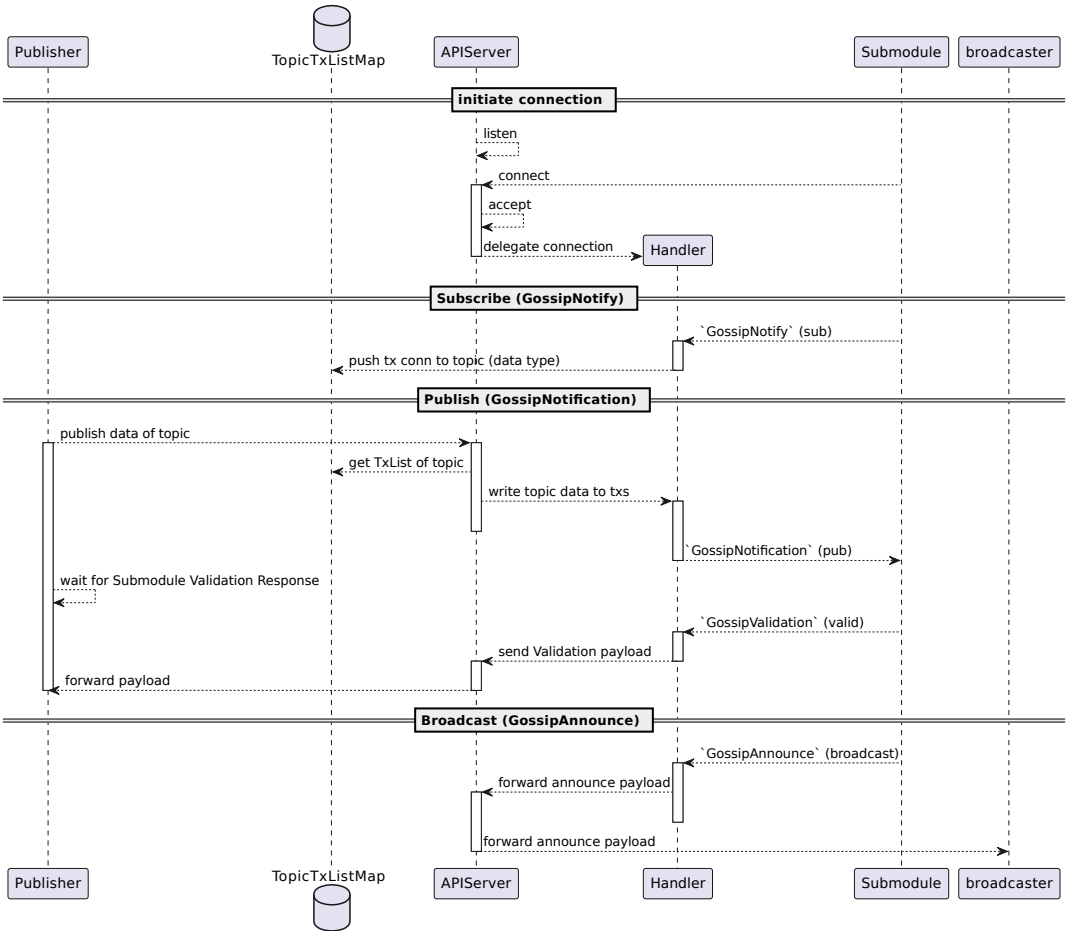

Figure 2.2: API Communication Sequence Diagram

the response of the RPS module to connect to peers and spread information between these peers. The P2P module utilizes two main structures to organize its tasks: Server and Peer. The server exposes the high level APIs like connect and send to other `GOSSIP` submodules. Internally, the server stores all currently active peer connections with their respective states in a key-value store. Identities, hashed public key and public IP address pairs, as discussed in Section 2.3, are used as keys for this map. Each map value has a reference to the corresponding Peer instance. The Peer instance represents the current state of the connection and exposes methods for communication with the associated peer over the network. The communication initiation process is outlined further in the Section 2.3. Additionally, the peer holds a reference to the underlying TCP socket, peer identity and public key of the respective peer. The server uses asynchronous tasks to listen for incoming messages from all active peers in parallel. Upon message reception, from one of the peers, the decoded message, peer identity and peer address are sent to the server receive channel. The message format is discussed in Section 2.3. The receive channel is then read from asynchronously and the result is forwarded to the broadcaster submodule.
If the message is found to be valid and the TTL value is not equal to one, it is forwarded to other connected peers. Messages with TTL of one are considered finished and are not forwarded further. If a peer is not connected to a sufficient number of peers, the message is kept in a stack for further transmission and will be further propagated when new peers become available. Additionally, the received message is trans-coded into the API-Communication format and forwarded to the Publisher submodule for further spread to other internal modules as described in Section 2.2.1.

2.2.2 Process Architecture

As the Gossip Module is a network application, the module uses the asynchronous runtime Tokio [3] to achieve fast, reliable, and scalable execution. This usage of this runtime defines the module's process architecture as asynchronous, event-driven, and non-blocking. In particular, the Tokio runtime launches asynchronous units of operation called *tasks*, which a task scheduler distributes to a set of core threads spawned upon the runtime's execution. These core threads concurrently execute assigned tasks until the tasks yield, i.e., when a task can not make further progress. For instance, a task which awaits an I/O operation may yield to allow the thread to execute a different task. Thereupon, the scheduler swaps the tasks for one which is ready for further execution.

Figure 2.2 applies this process architecture to ensure concurrent execution. The participants of this figure correspond for the most part to design-pattern actors [4], as they mainly communicate by passing messages instead of sharing memory. The tasks thereby implement these actors, allowing the actors to execute independently of

each other. This applies to the API Server, which communicates with the Publisher and Broadcaster with message passing via channels. Upon accepting a connection, the API Server spawns a task for a Handler actor, which manages the created connection. Further, the API Server and the Handler also communicate via message passing. The Handler thereby passes messages read from the connection to the API Server for further processing. Only the shared topic-connection-reference key-value store breaks this principle, as the actors share this memory to avoid introducing the additional complexity of a database handler actor.

2.2.3 Networking Model

As described in the previous Section 2.2.2, the Gossip module uses an asynchronous networking model. As Gossip is a networking module used by the node's internal submodules as well as other peers, it must perform concurrent networking. Hence, this module performs many I/O operations, which would require a synchronous program to frequently block thread executions to wait for events. As this waiting wastes CPU cycles which can otherwise be used to perform further operations, the Gossip module employs asynchronous networking to bridge these gaps, as described in the previous Section 2.2.2.

Again, Figure 2.2 shows how Gossip uses this networking paradigm to avoid blocking threads. When a connection Handler writes a `GOSSIP NOTIFICATION` message to its connection, it must subsequently wait and return a `GOSSIP VALIDATION` message of the corresponding module. In a synchronous implementation, the Gossip module would have to block the thread executing the Handler until the `GOSSIP VALIDATION` message arrives, wasting many CPU cycles. Yet, in Gossip module's asynchronous implementation, the Handler task yields its execution upon waiting for the corresponding module's response. This yielding of execution allows the responsible thread to instead for instance execute the API Server to handle new incoming connections.

2.3 Peer-to-Peer Protocol

The Peer-To-Peer Protocol describes the communication between two individual peers. All P2P connections use the TCP for data transmission. Protocol Buffers are used for serialization and deserialization. Before a connection is considered ready, both parties must exchange public keys and complete a Proof-Of-Work (PoW) challenge. The challenge mechanism is currently analogous to the one described in Section "4.1. Group registration" of the specification. The SHA2-256 hash is replaced by BLAKE3 hash. Additionally, we require both parties to perform this PoW challenge process before treating a connection as active. BLAKE3 is a Merkel Tree based hashing algorithm [5].

BLAKE3 is an evolution of BLAKE2, which was considered as a candidate for SHA3. Compared to SHA2, BLAKE3 is designed to be faster on a wide range of hardware and is furthermore not susceptible to length extension attacks. We initially allow the connecting party to set the difficulty of the challenge in the initial request to tune our PoW setup to work correctly with BLAKE3. This mechanism will be removed in the final version. After a connection is open, payload data can be transferred over the connection. Both parties store the public keys of the opposing peer for later reference in a cache. In future iterations, the public keys may be used to ensure privacy between the two parties. Further, future security measures are outlined in Section 2.4.
The identity of a node is defined as the BLAKE3 hash of the public key and the public IP address of the respective peer. The identities of both parties are used as fields in the Connection Challenge Response message and act as a countermeasure to Sybil attacks by enforcing PoW on a per-connection basis.

2.3.1 Message Format

This section outlines the individual message types currently used in the P2P protocol. The message figures in this section are symbolic and should be treated as a rough overview. Actual wire formats may differ due to Protocol Buffers encoding.

Connection Challenge

Connection Challenge messages fulfill two roles, public key exchange and protection against Sybil attacks by enforcing nodes to perform work before being accepted by their peers. Both parties initially exchange Connection Challenge messages. The connecting party must directly send out the challenge message first, the receiving party must send its connection challenge in direct response. Each connection challenge message also includes the identity and public key of the sending party. The difficulty dictates the length of the zero prefix in the hash of the challenge response message.

Src. Identity: 256bit
Identity of the sending party.

Signature
Signature of the Src. Identity field using the senders private key.

Challenge: 32bit
Randomly selected challenge sequence that must be included in response.

Difficulty: 8bit
Length of zero prefix in response message hash.

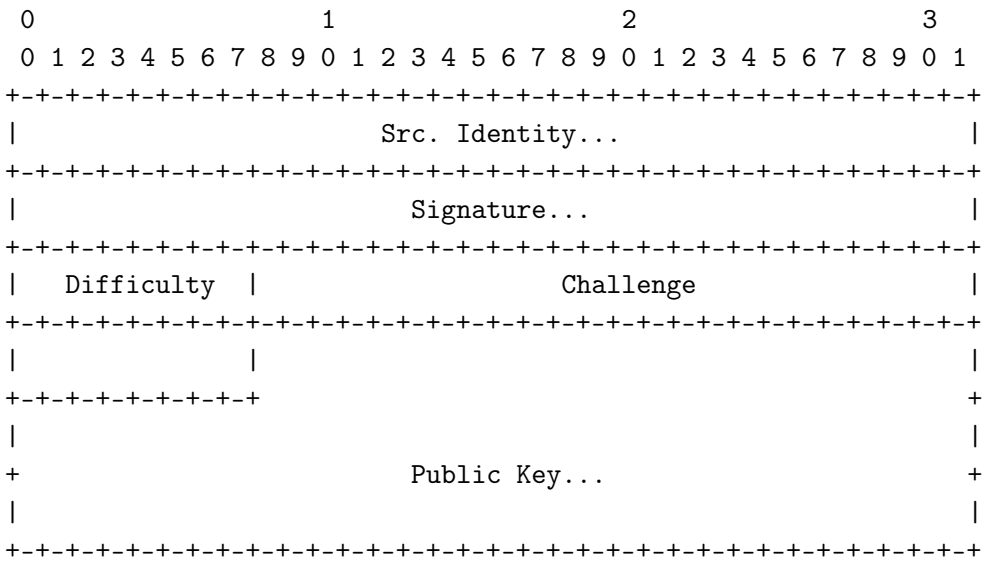

Figure 2.3: Connection Request

Public Key
Asymmetric public key of the sending party.

Connection Challenge Response

This message is sent by both parties as a response to the Connection Challenge message after a solution was computed. If the solution was wrong or too slow, a Connection Error message is transmitted, and the connection process must be cancelled. In further iterations of the P2P protocol this message will be encrypted.

Src. Identity: 256bit
Identity of the sending party.

Dst. Identity: 256bit
Identity of the receiving party.

Challenge: 32bit
Randomly selected challenge sequence that must be included in response.

Nonce: 32bit
The nonce is any value that will lead to the message encoding to solve the challenge.

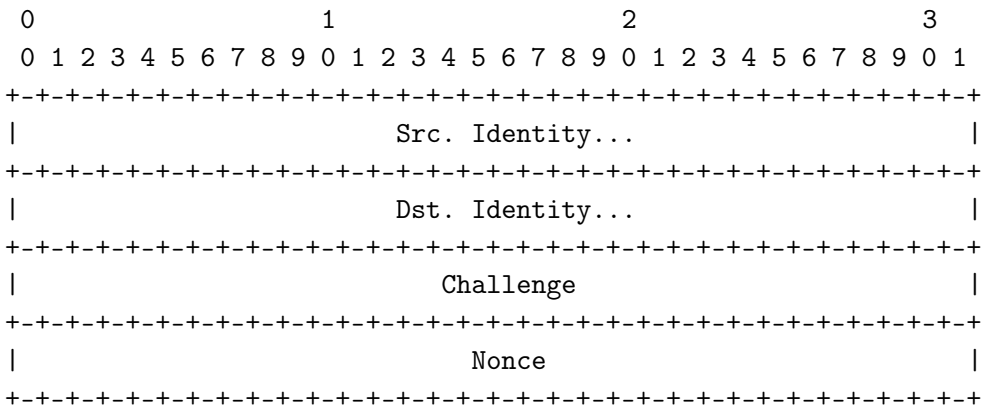

Figure 2.4: Connection Request Response

Connection Okay

The Connection Okay message is transmitted upon the reception of a valid Connection Challenge Response. If both parties have transmitted a Connection Okay message, the connection is active and can be used for bidirectional data transmission. This message has no further fields or payload.

Payload

The Payload message is used to transfer all non-connection-specific messages over the connection. This message type must only be sent after the connection process was completed successfully.

TTL: 8bit
Time To Live, represents the number of hops this message must be forwarded to.

Reserved: 8bit
Reserved field of the specification protocol, currently included for future compatibility and may be removed for the final version.

Data Type: 8bit
Data Type as defined in the specification and analogous to the `GOSSIP_ANNOUNCE` data type.

Payload Size: 16bit
The length of the payload section.

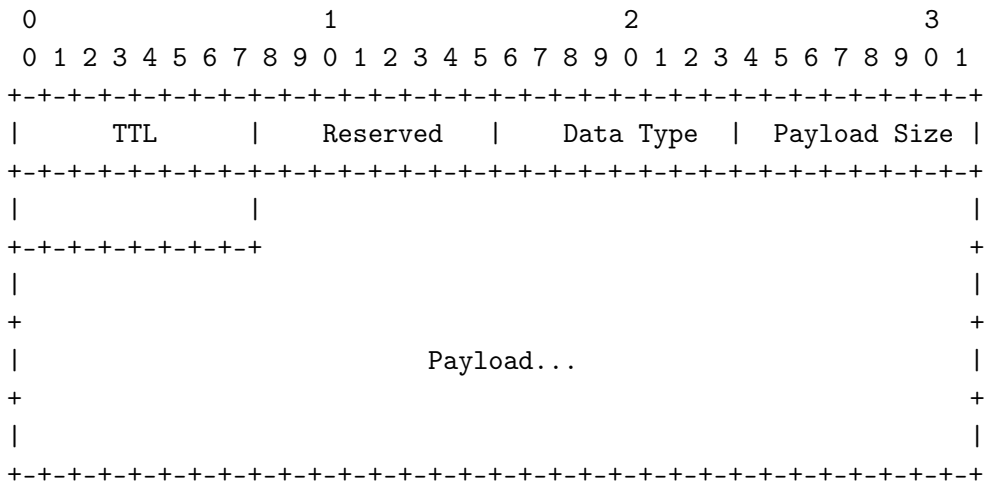

Figure 2.5: Payload Message

Payload: variable length
Data transmitted in the message.

Error

Error messages are transferred to communicate errors in the communication. Initially, only errors for connection establishment are defined.

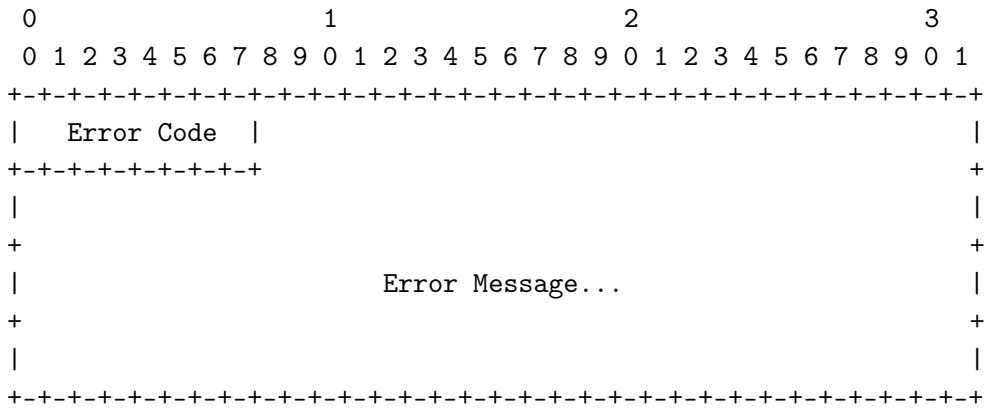

Figure 2.6: Error Message

Error Code: 8bit
Code representing the error.

Error Message: variable length
Error message description encoded as variable length UTF-8 string.

Name	Error Code.	Hex Encoding
Undefined	00000000	0x00
Invalid Request	00000001	0x01
Checksum Invalid	00000010	0x02
Response Timeout	00000011	0x03
Reserved	00000100	0x04
Reserved	...	
Reserved	11111111	0xFF

Table 2.1: Currently, defined Error Codes

2.4 Future Work

- Broadcaster submodule
  The broadcaster submodule has yet to be written
- Configuration loading
  A submodule for configuration loading and parsing must be created
- API-Communication Integration
  We must integrate the API-Communication submodule into the Broadcaster submodule
- Publisher Integration
  We must integrate the Publisher submodule into the Broadcaster submodule as well

2.4.1 P2P-Communication

- Finish switch to TCP
  We are currently still refactoring the P2P module to work with TCP

- Proof-Of-Work
  PoW has to be implemented.
- Switch to TCP Framing
  Currently, TCP packets are manually de- and encoded upon reception and transmission. We want to switch to a TCPStream framer to unify this process.
- Testing
  The current tests must be updated to work with the new P2P-Communication submodule
- Security (Confidentiality and Authenticity)
  The current P2P communication does not enforce confidentiality or authenticity between two parties. This allows an attacking party to easily manipulate the communication, for example with a Man In The Middle attack. Additionally, the communication is currently not encrypted in any way and can be eavesdropped easily. To combat this, we intend to wrap the P2P connections with a encryption layer. The specific cryptography combination is currently under active discussion, we are leaning towards a rust wrapper for libsodium. This library offers safe out of the box encryption mechanisms, limiting the area for errors in our implementation.

2.4.2 Testing

- P2P-Communication tests
  The current tests must be updated to work with the new P2P-Communication submodule.
- Voidphone tests
  Tests for the general system operation must be integrated.
- Module interoperability tests
  Test compatibility with other non gossip modules.

2.5 Workload Distribution

For the workload distribution, we refer to [1]. The therein described distribution still holds. Further, Section 2.4 states what work the group members must still execute.

2.6 Effort Spent for the Project

This section describes the total effort invested in this project per group member so far. We measure this effort in hours. Further, the corresponding work consists of all tasks involved in building the Gossip module, for instance software development, reading of documentation, group meetings, and report writing. For each group member, this effort was recorded using time-tracking software.

In the case of the group member Robert Schambach, this effort amounts to 35-45 hours spent on the project.

For Wladislaw Meixner the effort amounted to 30-40 hours. In addition to implementation and design, a substantial portion of this time was spent on learning key rust and Tokio concepts required to implement the proposed solution.

Bibliography

[1] W. M. Robert Schambach. (Jun. 10, 2022). "Initial report," [Online]. Available: https://gitlab.lrz.de/netintum/teaching/p2psec_projects_2022/Gossip-4/-/blob/main/docs/initial-report.pdf (visited on 07/06/2022).

[2] S. H. Totakura, L. Schwaighofer, R. von Seck, and G. Carle. (May 2, 2022). "VoidPhone project specification," [Online]. Available: https://www.moodle.tum.de/pluginfile.php/3690959/mod_resource/content/1/specification.pdf (visited on 05/03/2022).

[3] *Tokio*, original-date: 2016-09-09T22:31:36Z, Jul. 9, 2022.

[4] C. Hewitt, P. Bishop, and R. Steiger, "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence," in *Advance Papers of the Conference*, Stanford Research Institute Menlo Park, CA, vol. 3, 1973, p. 235.

[5] S. N. Jack O'Connor Jean-Philippe Aumasson. (Jul. 11, 2022). "Blake3 one function fast everywhere," [Online]. Available: https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf (visited on 07/11/2022).